Correspondence and Independence of Numerical Evaluations of Algorithmic Information Measures[*]

Fernando Soler-Toscano[1], Hector Zenil[2], Jean-Paul Delahaye[3] and Nicolas Gauvrit[4]

[1] Grupo de Lógica, Lenguaje e Información, Universidad de Sevilla, Spain.

[2] Department of Computer Science, University of Sheffield, UK.

[3] Laboratoire d'Informatique Fondamentale de Lille, France.

[4] LDAR, Université de Paris VII, Paris, France.

Abstract

We show that the Kolmogorov-Chaitin complexity $K(s)$ of a string s numerically approximated using Levin's coding theorem with the associated measure $K_m(s)$ (a method we have therefore named the *Coding Theorem Method*) as calculated from the frequency of production of a large set of small deterministic Turing machines with up to 5 states (and 2 symbols) correlates with the number of instructions used by the Turing machines, in agreement with strict integer-value program-size complexity. Nevertheless, K_m proves to be a finer-grained measure and a better approach for distinguishing K for short strings from non-integer evaluations. We also show that neither K_m nor the number of instructions used suggests any correlation with Bennett's concept of Logical Depth (the shortest runtime of the shortest computer program producing s). And we announce a first version of an online complexity calculator based on a combination of theoretical concepts as an implementation of our *Coding Theorem Method*.

Keywords: Coding Theorem Method; Kolmogorov-Chaitin complexity; Solomonoff-Levin algorithmic probability; Levin's universal distribution; Levin-Chaitin coding theorem; Program-size complexity; small Turing machines.

[*]The webpage of the Algorithmic Nature group is http://www.algorithmicnature.org. An online complexity calculator is provided in http://www.complexitycalculator.com.

1 Introduction

Kolmogorov complexity (also known as Kolmogorov-Chaitin, algorithmic or program-size complexity) is recognized as a fundamental concept, but it is also often thought of as having little or no applicability because it is not possible to provide stable numerical approximations for finite–particularly short–strings by using the traditional approach, namely lossless compression algorithms. We advance a method that can overcome this limitation, and which even if limited in ways both theoretical and practical, nonetheless offers a means of providing sensible values for the complexity of short strings, complementing the traditional lossless compression method that works well for long strings. This is done at the cost of massive calculations in order to use the coding theorem from algorithmic probability that relates the frequency of production of a string with its Kolmogorov complexity.

Bennett's logical depth, on the other hand, is a measure of the complexity of strings that, unlike Kolmogorov complexity, measures the *organized* information content of a string and not its random *incompressible* complexity. To calculate the non-computable is always a challenge, and it can of course only be achieved partially, but Bennett's logical depth is even more difficult to calculate given its own particularities. This approach, however, represents a first numerical attempt to provide *exact* calculations for a fixed formalism.

The independence of the two measures, that is, Kolmogorov complexity and Logical depth (which has been established theoretically), is numerically tested and reported in this paper. Our work is in agreement with what the theory predicts, even for short strings and despite the limitations of this approach. The ability to apply these concepts to practical problems (established in a series of articles (see [13, 22])) is novel, and they should prove to have many applications where evaluations of the complexity of finite short strings are needed. Their practical numerical approximations suggest that our Coding Theorem Method is sound and of potential use where compression algorithms—the method traditionally used—fail to provide any sensible approximation to K.

In Sections 2, 3, 6 and 5 we introduce the measures, tools and formalism used in Section 7 to advance our Coding Theorem Method to approximate the Kolmogorov complexity of short strings. Finally in Section 7, we report the results of the analysis of the comparison among the various measures calculated by the method, particularly the complexity by number of instructions and the logical depth.

2 Kolmogorov-Chaitin complexity

When researchers have chosen to apply the theory of algorithmic information (AIT), which in principle is not supposed to be of any practical use [7], it has proven to be of great value, for example, for DNA false positive repeat sequence detection in genetic sequence analysis [19], in distance measures and classification methods [8], and in numerous other applications [18]. This effort has, however, been hamstrung by the limitations of compression algorithms–currently the only method used to approximate the Kolmogorov complexity of a string–given that this measure is not computable.

Central to AIT is the definition of algorithmic (Kolmogorov-Chaitin or program-size) complexity [16, 6]:

$$K_T(s) = \min\{|p|, T(p) = s\} \tag{1}$$

That is, the length of the shortest program p that outputs the string s running on a universal Turing machine T. A technical inconvenience of K as a function taking s to the length of the shortest program that produces s is its uncomputability. In other words, there is no program which takes a string s as input and produces the integer $K(s)$ as output. This is usually considered a major problem, but one ought to expect a universal measure of complexity to have such a downside.

The measure was first conceived to define randomness and is today the accepted objective mathematical measure of finite randomness, among other reasons because it has been proven to be mathematically robust (by virtue of the fact that several independent definitions converge in it).

A classic example is a string composed of an alternation of bits, such as $(01)^n$, that can be described as "n repetitions of 01". The string can grow fast while the description will only grow by about $\log_2(n)$. On the contrary, a random-looking string such as 011001011010110101 may not have a much shorter description than itself.

Traditionally, the way to approach the algorithmic complexity of a string has been by using lossless compression algorithms. The result of a lossless compression algorithm is an upper bound of its algorithmic complexity. Short strings, however, are difficult to compress in practice, and the theory does not provide a satisfactory solution to the problem of the instability of the measure for short strings.

The invariance theorem guarantees that complexity values will only diverge by a constant c (e.g. the length of a compiler, a translation program between U_1 and U_2) and that hey will converge at the limit.

Invariance Theorem ([4, 18]): If U_1 and U_2 are two universal Turing machines and $K_{U_1}(s)$ and $K_{U_2}(s)$ the algorithmic complexity of s for U_1 and U_2, there exists a constant c such that:

$$|K_{U_1}(s) - K_{U_2}(s)| < c \tag{2}$$

Hence the longer the string, the less important c is (i.e. the choice of programming language or universal Turing machine). However, in practice c can be arbitrarily large, thus having a very great impact on short strings.

3 Bennett's Logical Depth

The concept of Kolmogorov-Chaitin complexity formalizes the concepts of simplicity and randomness by means of information. As mentioned before, several applications based on K using compression algorithms have been successfully developed to date. None, however, seems to have exploited the concept of logical depth, with the exception of a previous paper of ours [22], which did so with encouraging results. A measure of the complexity of a string can be arrived at by combining the notions of algorithmic information content and time complexity. According to the concept of logical depth [1, 2], the complexity of a string is best defined by the time that an unfolding process takes to reproduce the string from its shortest description. The longer it takes, the more complex. Hence complex objects are those which can be seen as "containing internal evidence of a nontrivial causal history." The concept of logical depth takes into account the plausible history of an object as an unfolding phenomenon. It combines the concept of the shortest possible description of an object with the time that it takes to evolve to the state it is in at any given moment.

A typical example that illustrates the concept of logical depth and its characterization as a measure of physical complexity is a sequence of fair coin tosses. Such a sequence would have a high information content (algorithmic complexity) because the outcomes are random, but little value (logical depth) because they are easily generated and carry no message, no meaning. The string 1111...1111 is also logically shallow. Its minimal program, while very small, requires little time to evaluate. In contrast, the binary representation of the number π is not shallow, because although it is highly compressible (by any known formula producing π), the generating algorithms require computational time to produce several digits of its expansion. A better example is Chaitin's Ω number [6], the digits of which encode the halting probability of a universal Turing machine, and which is known to be very deep since no computable process can expand Ω except for a finite number of digits [4, 14].

Bennett offers a careful development [1] of the notion of logical depth, taking into account near-shortest programs as well as the shortest one, hence the significance value for a reasonably robust and machine-independent measure. Algorithmic complexity and logical depth are intimately related theoretically, because logical depth requires an approximation of algorithmic complexity. But they are also supposed to be different measures. Unlike algorithmic complexity, which assigns a high complexity to both random and highly organized objects, placing them at the same level, logical depth assigns a low complexity to both random and trivial objects. It is thus more in keeping with our intuitive sense of the complexity of physical objects, because trivial and random objects are intuitively easy to produce, do not have a long history, and unfold quickly. A clear, detailed explanation pointing out the convenience of the concept of logical depth as a measure of organized complexity as compared to plain algorithmic complexity, which is what is usually used, is provided in [10]. We show that it is the case that these measures measure different things and that they accord with our intuitive sense of what each is supposed to measure (randomness versus simplicity in the case of Kolmogorov complexity, and structure versus randomness and simplicity in the case of logical depth).

For finite strings, one of Bennett's formal approaches to the logical depth of a string is defined as follows:

Let s be a string and d a significance parameter. A string's depth at significance d is given by

$$LD_d(s) = \min\{T(p) : (|p| - |p'| < d) \text{ } and \text{ } (U(p) = s)\} \tag{3}$$

with $|p'|$ the length of the shortest program for s, (therefore $K(s)$). In other words, $LD_d(s)$ is the least time T required to compute s from a d-incompressible program p on a Turing machine U.

Each of the three linked definitions of logical depth provided in [1] comes closer to a definition in which near-shortest programs are taken into consideration. In this experimental approach we make no such distinction among significance parameters, so we will denote the logical depth of a string s simply by $LD(s)$.

Like $K(s)$, $LD(s)$ as a function of s is uncomputable. A novel feature of this research is that we were able to provide exact approximations of logical depth. This was achieved by running Turing machines sorted by increasing size and finding the smallest and fastest machine from among a relatively large sample of Turing machines that produced a given string.

4 Solomonoff-Levin Algorithmic Probability

The algorithmic probability (also known as Levin's semi-measure) of a string s is a measure that describes the expected probability of a random program p running on a universal (prefix-free[1]) Turing machine T producing s. Formally [21, 17, 6],

$$m(s) = \Sigma_{p:T(p)=s} 1/2^{|p|} \tag{4}$$

i.e. the sum over all the programs for which T with p outputs s and halts.

Levin's semi-measure[2] $m(s)$ defines a distribution known as the Universal Distribution [15]. It is important to notice that the value of $m(s)$ is dominated by the length of the smallest program p (when the denominator is larger). The length of the smallest p that produces the string s is, however, $K(s)$. The semi-measure $m(s)$ is therefore also uncomputable, because for every s, $m(s)$ requires the calculation of $2^{-K(s)}$, involving K, which is itself uncomputable. An alternative [13] to the traditional use of compression algorithms is the use of the concept of algorithmic probability to calculate $K(s)$ by means of the following theorem.

Coding Theorem (Levin [17]):

$$|-\log_2 m(s) - K(s)| < c \tag{5}$$

An informal interpretation is that if a string has many long descriptions it also has a short one. It beautifully connects frequency to complexity, more specifically the frequency (or probability) of occurrence of a string with its algorithmic (Kolmogorov) complexity. The coding theorem implies that [9, 4] one can calculate the Kolmogorov complexity of a string from its frequency [12, 11, 23, 13], simply rewriting the formula as:

$$K_m(s) = -\log_2 m(s) + O(1) \tag{6}$$

[1]The group of valid programs forms a prefix-free set (no element is a prefix of any other, a property necessary to keep $0 < m(s) < 1$.) For details see [4]).

[2]It is called a *semi* measure because the sum is never 1, unlike probability measures. This is due to the Turing machines that never halt.

An important property of m as a semi-measure is that it dominates any other effective semi-measure μ, because there is a constant c_μ such that for all s, $m(s) \geq c_\mu \mu(s)$. For this reason $m(s)$ is often called a *universal distribution* [15].

5 Deterministic Turing machines

The ability of a universal Turing machine to simulate any algorithmic process[3] has motivated and justified the use of universal Turing machines as the language framework within which definitions and properties of mathematical objects are given and studied.

However, it is important to describe the formalism of a Turing machine, because exact values of algorithmic probability for short strings will be approximated under this model, both for $K(s)$ through $m(s)$ (denoted by K_m), and for $K(s)$ in terms of the number of instructions used by the smallest Turing machine producing s.

Consider a Turing machine with the binary alphabet $\Sigma = \{0, 1\}$ and n states $\{1, 2, \ldots n\}$ and an additional Halt state denoted by 0 (as defined by Rado in his original Busy Beaver paper [20]).

The machine runs on a 2-way unbounded tape. At each step:

1. the machine's current "state" (instruction); and
2. the tape symbol the machine's head is scanning

define each of the following:

1. a unique symbol to write (the machine can overwrite a 1 on a 0, a 0 on a 1, a 1 on a 1, and a 0 on a 0);
2. a direction to move in: -1 (left), 1 (right) or 0 (none, when halting); and
3. a state to transition into (which may be the same as the one it was in).

The machine halts if and when it reaches the special halt state 0. There are $(4n + 2)^{2n}$ Turing machines with n states and 2 symbols according to the formalism described above. The output string is taken from the number

[3]Under Church's hypothesis.

of contiguous cells on the tape the head of the halting n-state machine has gone through. A Turing machine is considered to produce an output string only if it halts. The output is what the machine has written on the tape.

6 The *Coding Theorem Method*

One can attempt to approximate $m(s)$ by running every Turing machine following a particular enumeration. A natural one is a quasi-lexicographical ordering, from shorter to longer by number of states and symbols. Let (n, m) be the set of Turing machines with n states and m symbols. It is clear that in this fashion once a machine produces s for the first time, one can directly calculate an exact value of K. Because this is the length of the first Turing machine in the enumeration of programs of increasing size that produces s, there is no shorter machine producing s, and from Turing universality we know there is a machine $T \in (n, m)$ that produces s.

Let's now formalize a function $D(n, m)$ as was previously done in [13], as an approximation of $m(s)$ for binary strings as follows:

$$D(n,m)(s) = \frac{|\{T \in (n,m) : T(p) = s\}|}{|\{T \in (n,m) : T \textit{ halts } \}|} \tag{7}$$

Where $T(p)$ is the Turing machine with number p (and empty input) that produces s upon halting, and $|A|$ is, in this case, the cardinality of the set A. We have previously proved [23, 13] that the function $(n, m) \to D(n, m)$ is non-computable by reduction to the halting problem. However, $D(n, m)$ is lower semi-computable, meaning it can be computably approximated from below, for example, by running small Turing machines for which values of the Busy Beaver problem [20] are known. For example, for $n = 4$, the Busy Beaver function for maximum runtime S, tells us that $S(4, 2) = 107$ [3], so we know that a machine running on a blank tape will never halt if it hasn't halted after 107 steps, and we can therefore stop it manually.

Previously [23, 13] we had calculated the full output distribution of Turing machines with 2-symbols and $n = 1, ..., 4$ states for which the Busy Beaver values are known, in order to determine the halting time. That is a total of 36, 10 000, 7 529 536 and 11 019 960 576 Turing machines respectively.

Because there are a large enough number of machines to run even for a small number of machine states (n), applying the coding theorem provides a fine and increasingly stable (due to the invariance theorem) evaluation of $K(s)$ based on the frequency of production of a large number of Turing machines. But the number of Turing machines grows exponentially, and producing $D(5,2)$ requires considerable computational resources. Calculating $D(5,2)$ is an improvement on our previous numerical evaluations and provides a larger data set to work with and to draw more significant statistical conclusions from for the purposes of this research. There are 26 559 922 791 424 Turing machines with 5 states and 2 symbols, and the values of Busy Beaver functions for these machines are unknown. In what follows we describe how we proceeded. From now on, $D(n)$ with a single parameter will mean $D(n,2)$.

6.1 Reduction techniques

We did not run all the Turing machines with 5 states to produce $D(5)$ because one can take advantage of symmetries and anticipate some of the behavior of the Turing machines directly from their transition tables without actually running them (this is impossible in general due to the halting problem). We avoided some trivial machines whose results we know without having to run them. For example, machines with the initial transition moving to the halting state produce strings "0" and "1". It's easy to quantify these machines and avoid running them, saving the time that would be used generating trivial machines. Also, machines with the initial transition staying in the initial state will not halt (as they run on a blank tape); they always remain in the initial state. So we are interested in machines with the initial transition moving to a state different from both the initial and the halting states. Moreover, we can exploit the left-right symmetry and run only machines starting on the right, and for every string s produced in the output data, include the reverse of s with the same (or an increased) frequency.

To restrict the generating machines by imposing these constraints on them we created a reduced enumeration that for n states contains $2(n-1)(4n+2)^{2n-1}$ machines, with the initial transition moving to the right to a state different from both the initial and halting states.

For $n=5$ this means running only 4/11 of the total number of Turing machines. Moreover, we need the output of those machines starting with a 0-filled tape and with a 1-filled tape. But we do not run any machine twice, as for every machine M producing the binary string s starting with a 1-filled tape, there is a 0-1 symmetric machine M' that when starting with a 0-filled tape produces the complement to one of s, that is, the result of replacing all 0s in s with 1s and all 1s with 0s.

After running the 9 658 153 742 machines in the reduced enumeration for $D(5)$, we completed the strings generated using the symmetries described.

We also counted the number of non-halting machines that were skipped.

6.2 Detecting non-halting machines

It is useful to avoid running machines that we can easily determine will not stop. These machines will consume the runtime without yielding an output. As we have shown above, we can avoid generating many non-halting machines. In other cases, we can detect them at runtime, by setting appropriate filters. The theoretical limit of the filters is the halting problem, which means that they cannot be exhaustive. But a practical limit is imposed by the difficulty of checking some filters, which takes up more time than the runtime that is saved.

We have employed some filters that have proven to be useful. Briefly, these are:

- **Machines without transitions to the halting state**. While the transition table is being filled, the simulator checks to ascertain whether there is some transition to the halting state. If not, it avoids running it.

- **Escapees**. These are machines that at some stage begin running forever in the same direction. As they are always reading new blank symbols, as soon as the number of non-previously visited positions is greater than the number of states, we know that they will not stop.

- **Cycles of period two**. These cycles are easy to detect. They are produced when in steps s and $s+2$ the tape is identical and the machine is in the same state and the same position. When this is the case, the cycle will be repeated infinitely.

These filters were implemented in our C++ simulator, which also uses the reduced enumeration of Section 6.1. To test them we calculated $D(4)$ with the simulator and compared the output to the list that was computed in [13], arriving at exactly the same results, and thereby validating our reduction techniques.

Running $D(4)$ without reducing the enumeration or detecting non-halting machines took 952 minutes. Running the reduced enumeration with non-halting detectors took 226 minutes.

6.3 Setting the runtime

The Busy Beaver for Turing machines with 4 states is known to be 107 steps [3], that is, any Turing machine with 2 symbols and 4 states running longer than 107 steps will never halt. However, the exact number is not known for Turing machines with 2 symbols and 5 states, although it is believed to be 47 176 870, as there is a candidate machine that runs for this length of time and halts and no machine with a greater runtime has yet been found.

So we decided to let the machines with 5 states run for 4.6 times the Busy Beaver value for 4-state Turing machines (for 107 steps), knowing that this would constitute a sample significant enough to capture the behavior of Turing machines with 5 states. The chosen runtime was rounded to 500 steps, which was used to construct the output frequency distribution for $D(5)$.

Not all 5-state Turing machines have been used to build $D(5)$, since only the output of machines that halted at or before 500 steps was taken into consideration. As an experiment to ascertain how many machines we were leaving out, we ran 1.23×10^{10} random Turing machines for up to 5000 steps. Among these, only 50 machines halted after 500 steps and before 5000 (that is, less than 1.75164×10^{-8}, because in the reduced enumeration we don't include those machines that halt in one step or that we know won't halt before we generate them, so it's a smaller fraction), with the remaining 1 496 491 379 machines not halting at 5000 steps. As far as these are concerned–and given that the Busy Beaver values for 5 states are unknown–we do not know after how many steps they would eventually halt, if they ever do. According to the following analysis, our election of a runtime of 500 steps therefore provides a good estimation of $D(5)$.

The frequency of runtimes of (halting) Turing machines has theoretically been proven to drop exponentially [5], and our experiments are closer to the theoretically predicted behavior. To estimate the fraction of halting machines that were missed because Turing machines with 5 states were stopped after 500 steps, we hypothesize that the number of steps S a random halting machine needs before halting is an exponential random variable, defined by $\forall k \geq 1, P(S = k) \propto e^{-\lambda k}$. We do not have direct access to an evaluation of $P(S = k)$, since we only have data for those machines for which $S \leq 5000$. But we may compute an approximation of $P(S = k | S \leq 5000)$, $1 \leq k \leq 5000$, which is proportional to the desired distribution.

A non-linear regression using ordinary least-squares gives the approximation $P(S = k | S \leq 5000) = \alpha e^{-\lambda k}$ with $\alpha = 1.12$ and $\lambda = 0.793$. The residual sum-of-squares is 3.392×10^{-3}; the number of iterations with starting values $\alpha = 0.4$ and $\lambda = 0.25$ is nine. The model's λ is the same λ appearing in the general law $P(S = k)$, and may be used to estimate the number of machines we lose by using a 500 step cut-off point for running time: $P(k > 500) \approx e^{-500\lambda} \approx 6 \times 10^{-173}$. This estimate is far below the point where it could seriously impair our results: the less probable (non-impossible) string according to $D(5)$ has an observed probability of 1.13×10^{-9}.

Although this is only an estimate, it suggests that missed machines are few enough to be considered negligible.

7 Comparison of K_m with the number of instructions used and Logical Depth

We now study the relation of K_m with the minimal number of instructions used by a Turing machine producing a given string, and with Bennett's concept of logical depth. As expected, K_m shows a correlation with the number of instructions used but not with logical depth.

7.1 Relating K_m to the number of instructions used

First, we are interested in the relation of $K_m(s)$ to the minimal number of instructions that a Turing machine producing a string s uses. Machines in $D(5,2)$ have a transition table with 10 entries, corresponding to the different pairs $[n,m]$, with s one of the five states and m either "0" or "1". These are the 10 instructions that the machine can use. But for a fixed input not all instructions are necessarily used. Then, for a blank tape, not all machines that halt use the same number of instructions. The simplest cases are machines halting in just one step, that is, machines whose transition for $(init\_state, blank\_symbol)$ goes to the halting state, producing a string "0" or "1". So the simplest strings produced in $D(5,2)$ are computed by machines using just one instruction. We expected a correlation between the K_m-complexity of the strings and the number of instructions used. As we show, the following experiment confirmed this.

We used a sample of 2836×10^9 random machines in the reduced enumeration for $D(5,2)$, that is, 29% the total number of machines. The output of the sample returns the strings produced by halting machines together with the number of instructions used, the runtime and the instructions for the Turing machine. To save space, we only saved the smallest number of instructions found for each string produced, and the smallest runtime corresponding to that particular number of instructions.

After doing the appropriate symmetry completions we have 99 584 different strings, which is to say almost all the 99 608 strings found in $D(5,2)$. The number of instructions used goes from 1 to 10. When 1 instruction is

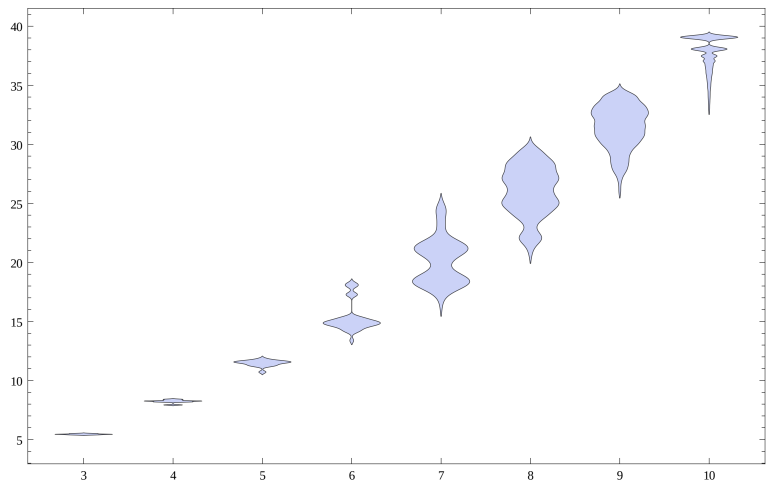

Figure 1: Distribution of K_m values according to the number of instructions used.

used only "0" and "1" are generated, with a K_m value of 2.51428. With 2 instructions, all 2-bit strings are generated, with a K_m value of 3.32744. For 3 or more instructions, Fig. 1 shows the distribution of values of K_m. Table 1 shows the mean K_m values for the different numbers of instructions used.

Used inst.	**Mean K_m**	**Mean Length**
1	2.51428	1
2	3.32744	2
3	5.44828	3
4	8.22809	4
5	11.4584	5
6	15.3018	6.17949
7	20.1167	7.76515
8	26.0095	9.99738
9	31.4463	12.6341
10	37.5827	17.3038

Table 1: Mean K_m and string length for different numbers of instructions used.

This accords with our expectations. Machines using a low number of instructions can be repeated many times by permuting the order of states.

So the probability of producing their strings is greater, which means low K_m values.

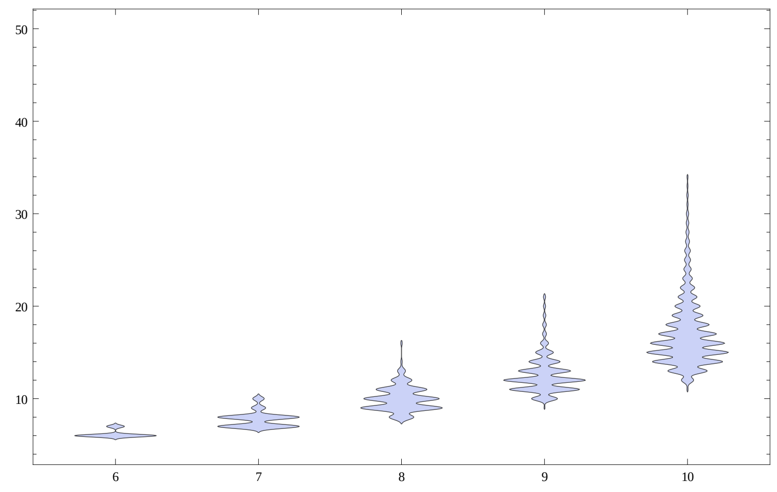

Figure 2: Instructions used and string lengths.

We can also look at the relation between the number of instructions used and the length of the strings produced. For $1 \leq i \leq 5$, all strings of length i are produced by machines using i instructions. For a greater number of instructions used, Fig. 2 shows the distribution of string lengths. Table 1 shows the mean length for each number of instructions used.

The correlation $r_{K_m,N} = 0.83$ is a good indicator for quantifying the apparent relation between K_m and the number N of instructions used, proving a strong positive link. However, since the length L of outputs is linked with both variables, the partial correlation $r_{K_m,N.L} = 0.81$ is a better index. This value indicates a strong relation between K_m and N, even while controlling for L.

7.2 Logical Depth and K_m

As explained above, we have also found that the machines which generate each string using the minimum number of instructions also have the minimum runtime. These runtimes are related to Bennett's logical depth (LD), as they are the shortest runtimes of the smallest Turing machines producing each string in $D(5,2)$.

We have partitioned the runtime space from 1 to 500 (our runtime bound) into 20 groups of equal length (25 steps). In order to explore the

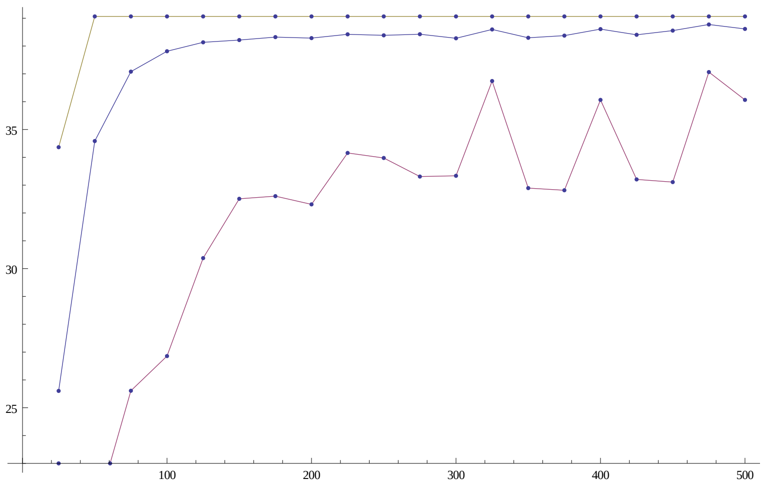

Figure 3: LD and K_m (min, mean and max values).

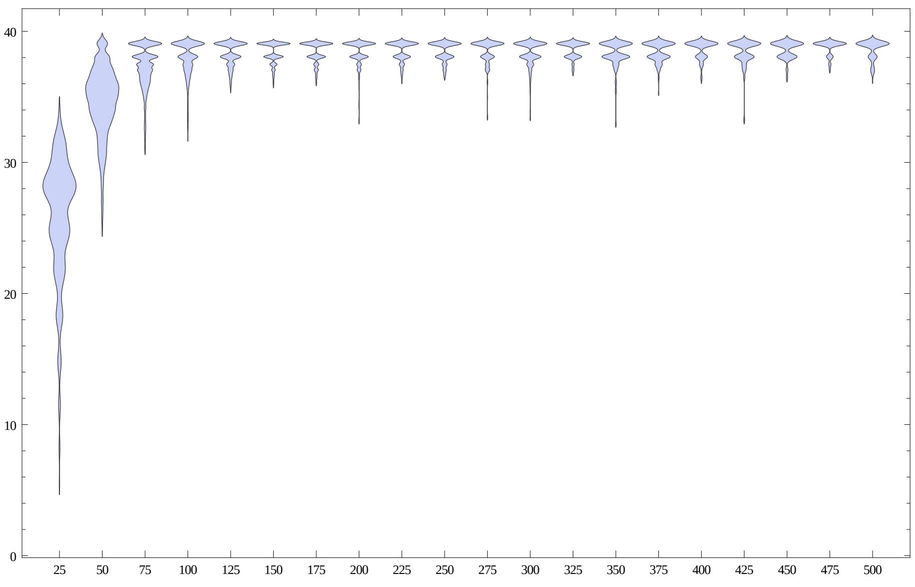

Figure 4: LD and K_m (distribution)

relation of K_m to Bennett's LD we are interested in the values of K_m for the strings in each group. Fig. 3 shows the minimum, mean and maximum K_m values for each of the runtime groups. The same information is in Table 1. The distribution of K_m values for the different groups is shown in Fig. 4. For each interval, the maximum runtime is shown on the horizontal axis.

Runtime	**Min** K_m	**Mean** K_m	**Max** K_m
1-25	2.51428	25.6049	34.3638
26-50	21.0749	34.5849	39.0642
51-75	25.6104	37.0796	39.0642
76-100	26.8569	37.8125	39.0642
101-125	30.3777	38.1337	39.0642
126-150	32.5096	38.2150	39.0642
151-175	32.6048	38.3208	39.0642
176-200	32.3093	38.2850	39.0642
201-225	34.1573	38.4213	39.0642
226-250	33.9767	38.3846	39.0642
251-275	33.3093	38.4249	39.0642
276-300	33.3363	38.2785	39.0642
301-325	36.7423	38.5963	39.0642
326-350	32.8943	38.2962	39.0642
351-375	32.8163	38.3742	39.0642
376-400	36.0642	38.6081	39.0642
401-425	33.2062	38.4035	39.0642
426-450	33.1100	38.5543	39.0642
451-475	37.0642	38.7741	39.0642
476-500	36.0642	38.6147	39.0642

Table 2: Extreme and mean K_m values for different runtime intervals.

We now provide some examples of the discordance between K_m and LD. "0011110001011" is a string with high K_m and low LD. Fig. 5 shows the transition table of the smallest machine found producing this string. The runtime is low–just 29 steps (of the 99 584 different strings found in our sample, only 3 360 are produced in fewer steps), but it uses 10 instructions and produces a string with complexity 39.0642. It is the greatest complexity we have calculated for K_m. Fig. 6 shows the execution of the machine.

On the other hand, "$(10)^{20}1$" is a string with high LD but low K_m value. Fig. 7 shows the transition table of the machine found producing this string, and Fig. 8 depicts the execution. The machine uses 9 instructions and runs

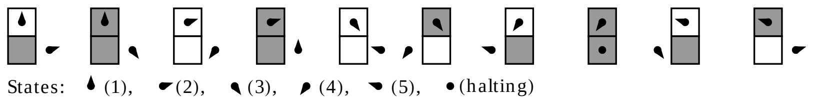

Figure 5: Transition table of a machine producing “0011110001011”.

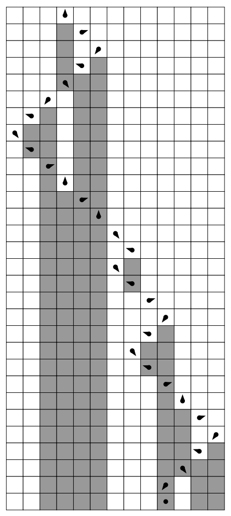

Figure 6: Execution of the machine producing “0011110001011”.

for 441 steps (only 710 strings out of the 99 584 strings in our sample require more time) but its K_m value is 33.11. This is a low complexity if we consider that in K_m there are 99 608 strings and that 90 842 are more complex than this one.

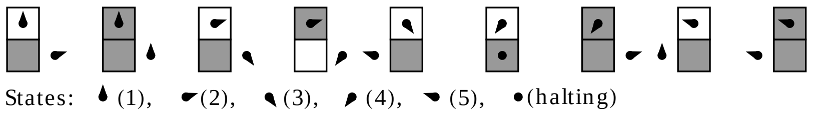

Figure 7: Transition table of a machine producing "$(10)^{20}1$".

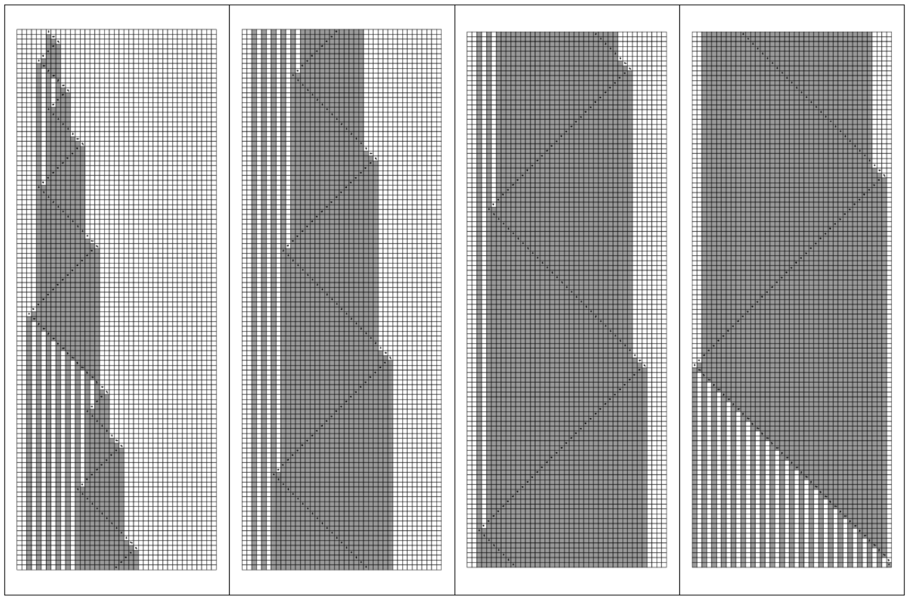

Figure 8: Execution of the machine producing "$(10)^{20}1$".

We may rate the overall strength of the relation between K_m and LD by the correlation $r_{K_m,LD} = 0.41$, corresponding to a medium positive link. As we previously mentioned however, the fact that the length L of the strings is linked with both variables may bias our interpretation. A more relevant measure is thus $r_{K_m,LD.L} = -0.06$, a negative value indicating no significant relation between K_m and LD once L is controlled.

8 Concluding remarks

As we expected, the Kolmogorov-Chaitin complexity evaluated by means of Levin's Coding Theorem from the output distribution of small Turing machines correlates with the number of instructions used but not with logical depth. Logical depth yields a reasonable measure of complexity that is different from the measure obtained by considering algorithmic complexity (K) alone, and this investigation proves that all these three measures (Kolmogorov-Chaitin Complexity, Solomonoff-Levin Algorithmic Probability and Bennett's Logic Depth) are numerically approachable, sound and consistent with theoretical expectations, and may be used in real-world applications. K as a measure of program size is supposed to be an integer (the length of a program in bits). K_m, however, yields non-integer values. Because K_m is shown to be a finer measure than the length of Turing machines, these results also justify the utility of non-integer values in the approximation of the algorithmic complexity of short strings, which also means being able to avoid the longer calculations that must be undertaken if only integer values were allowed.

An online tool that we have named the *Online Algorithmic Complexity Calculator* (or OACC) available at http://www.complexitycalculator.com is a long-term project to develop an online tool implementing the semi-computable measures of complexity described in this paper, and is expected to be expanded in the future.

It currently implements numerical approximations of Kolmogorov complexity and Levin's distribution for short binary strings following the numerical methods described herein, strings for which lossless compression algorithms fail to approximate their Kolmogorov complexity. Hence it provides a complementary and alternative method to compression algorithms.

The OACC is intended to provide a comprehensive framework of universal mathematical measures of randomness, structure and simplicity for researchers and professionals. It can be used to provide objective measures of complexity in a very wide range of disciplines, from bioinformatics to psychometrics, from linguistics to finance. More measures, more data and better approximations will be gradually incorporated in the future, covering a wider range of objects, such as longer binary strings, non-binary strings and n-dimensional arrays (such as images).

References

[1] C.H. Bennett, Logical Depth and Physical Complexity in Rolf Herken (ed) *The Universal Turing Machine–a Half-Century Survey,* Oxford University Press 227–257, 1988.

[2] C.H. Bennett, How to define complexity in physics and why. In *Complexity, entropy and the physics of information.* Zurek, W. H., Addison-Wesley, Eds. SFI studies in the sciences of complexity, p 137-148, 1990.

[3] A.H. Brady, The determination of the value of Rado's noncomputable function $Sigma(k)$ for four-state Turing machines, *Mathematics of Computation 40* (162): 647–665, 1983.

[4] C.S. Calude, *Information and Randomness*, Springer, 2002.

[5] C.S. Calude and M.A. Stay, Most Programs Stop Quickly or Never Halt, *Advances in Applied Mathematics*, 40, 295-308, 2008.

[6] G.J. Chaitin, On the length of programs for computing finite binary sequences: Statistical considerations, *Journal of the ACM*, 16(1):145–159, 1969.

[7] G.J. Chaitin. *From Philosophy to Program Size,* 8th. Estonian Winter School in Computer Science, Institute of Cybernetics, Tallinn, 2003.

[8] R. Cilibrasi, P. Vitanyi, Clustering by Compression, *IEEE Transactions On Information Theory,* 51, 4, 1523–1545, 2005.

[9] T.M. Cover and J.A. Thomas, *Information Theory,* J. Wiley and Sons, 2006.

[10] J.P. Delahaye, *Complexité aléatoire et complexité organisée,* Editions Quae, 2009.

[11] J-P. Delahaye, H. Zenil, Towards a stable definition of Kolmogorov-Chaitin complexity, arXiv:0804.3459, 2007.

[12] J-P. Delahaye and H. Zenil, On the Kolmogorov-Chaitin complexity for short sequences. In C. Calude (ed.), *Randomness and Complexity: From Leibniz to Chaitin*, World Scientific, 2007.

[13] J.-P. Delahaye & H. Zenil, Numerical Evaluation of the Complexity of Short Strings: A Glance Into the Innermost Structure of Algorithmic Randomness, *Applied Math. and Comp.*, 2012.

[14] R. Downey & D.R. Hirschfeldt, *Algorithmic Randomness and Complexity*, Springer, 2010.

[15] W. Kircher, M. Li, and P. Vitanyi, The Miraculous Universal Distribution, *The Mathematical Intelligencer,* 19:4, 7–15, 1997.

[16] A.N. Kolmogorov, Three approaches to the quantitative definition of information, *Problems of Information and Transmission*, 1(1):1–7, 1965.

[17] L. Levin, Laws of information conservation (non-growth) and aspects of the foundation of probability theory., *Problems in Form. Transmission* 10. 206—210, 1974.

[18] M. Li, P. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications,* Springer, 2008.

[19] É. Rivals, M. Dauchet, J.-P. Delahaye, O. Delgrange, Compression and genetic sequence analysis., *Biochimie*, 78, pp 315-322, 1996.

[20] T. Radó, On non-computable functions, *Bell System Technical Journal,* Vol. 41, No. 3, pp. 877–884, 1962.

[21] R.J. Solomonoff, A formal theory of inductive inference: Parts 1 and 2. *Information and Control*, 7:1–22 and 224–254, 1964.

[22] H. Zenil, J.-P. Delahaye and C. Gaucherel, Image Information Content Characterization and Classification by Physical Complexity, *Complexity*, vol. 17–3, pages 26–42, 2012.

[23] H. Zenil, Une approche expérimentale à la théorie algorithmique de la complexité, dissertation in fulfilment of the degree of Doctor in Computer Science (jury members: J.-P. Delahaye and C.S. Calude, G. Chaitin, S. Grigorieff, P. Mathieu and H. Zwirn), Université de Lille 1, 2011.